

Scalable Task-Oriented Parallelism for Structure Based Incomplete LU Factorization ¹

Xin Dong
College of Computer Science
Northeastern University
Boston, MA 02115 / USA
xindong@ccs.neu.edu

Gene Cooperman*
College of Computer Science
Northeastern University
Boston, MA 02115 / USA
gene@ccs.neu.edu

Abstract

$ILU(k)$ is an important preconditioner widely used in many linear algebra solvers for sparse matrices. Unfortunately, there is still no highly scalable parallel $ILU(k)$ algorithm. This paper presents the first such scalable algorithm. For example, the new algorithm achieves 50 times speedup with 80 nodes for general sparse matrices of dimension 160,000 that are diagonally dominant.

The algorithm assumes that each node has sufficient memory to hold the matrix. The parallelism is task-oriented. We present experimental results for $k = 1$ and $k = 2$, which are the most commonly used cases in the practical applications. The results are presented for three platforms: a departmental cluster with Gigabit Ethernet; a high-performance cluster using an InfiniBand interconnect; and a simulation of a Grid computation with two or three participating sites.

Keywords: $ILU(k)$, parallel computing, preconditioning, Gaussian elimination, task-oriented parallelism

I. INTRODUCTION

Incomplete LU factorization (ILU) has been found to be quite effective for preconditioning iterative sparse linear system solvers. In general, ILU algorithms are divided into two categories: threshold based methods ($ILUT$) and structure based methods ($ILU(k)$) [12]. Because $ILU(k)$ algorithm derives a hierarchy of ILU preconditioners, it is a popular choice for sequential implementations. Unfortunately, the lack of a scalable $ILU(k)$ preconditioner prevents its use in the parallel domain.

Iterative solvers for linear systems need efficient preconditioners since they need to repeatedly precondition intermediate matrices during succeeding iterations. As one scales to many processors, the poor parallelism of an $ILU(k)$ preconditioner will cause it to dominate the time of a linear algebra solver. For more than 15 years, researchers have been actively looking for a scalable implementation of parallel ILU [1], [6], [13], [14], [15], [16], [17], [26]. This work presents the first scalable parallel $ILU(k)$ preconditioner.

Already in 1991, Michael T. Heath, Esmond Ng and Barry W. Peyton [13, page 420] stated that “sparse matrix computations involve more complex algorithms, sophisticated data structures, and irregular memory reference patterns, making efficient implementations on novel architectures substantially more difficult to achieve...”. In 2000, D. Hysom and A. Pothen [15], [16] proposed

a potentially scalable algorithm for the restrictive case of well-partitionable matrices. In 2002, Benzi [1, page 446] pointed out that “ILU preconditioners are widely believed to be ill-suited for implementation on parallel computers with more than a few processors”. This is most likely the reason that among the current commonly used linear algebra libraries such as LAPACK, CLAPACK, PLAPACK, ScaLAPACK and PETSc, parallel $ILU(k)$ algorithm is still missing. In 2006, P. Hénon and Y. Saad [14] used a static hierarchical graph decomposition algorithm, related to well-partitionable matrices. For this case, they demonstrate scalability using four IBM power5 SMP nodes, each with 16 shared memory processors.

In contrast to all earlier methods, we present a scalable $ILU(k)$ algorithm based on task-oriented parallelism for general diagonally dominant sparse matrices. Diagonal dominance is a standard condition even for sequential $ILU(k)$ algorithm.

We demonstrate general scalability for 80 separate nodes using an Infiniband interconnect. We see nearly linear speedup for dimension up to 160,000, up to 80 processors, and for values of $k = 1$ and $k = 2$. The algorithm is also valid beyond $k = 2$.

Generally speaking, larger k leads to a better preconditioner in the sense that the number of iterations decreases. However, larger k increases the time for preconditioning and the requirement for storage. For a sequential preconditioner, if the time of preconditioning for larger k dominates the computation, the average performance no longer improves and sometimes even drop down. Under such circumstance, our parallel algorithm speeds up preconditioning and allows us to use larger k and achieve better result.

The key idea is to use task-oriented parallelism [2]. This implementation uses TOP-C [3], although any implementation of master-worker parallelism would suffice. We augment this model by allowing a “Worker” to broadcast to all nodes. The approach is inspired by the earlier use of task-oriented parallelism for Gaussian elimination [2].

The $ILU(k)$ computation has two parts: *symbolic factorization* (computation of levels) and *numeric factorization* (computation of matrix entries where the level criterion is satisfied). For symbolic factorization, the $k = 1$ case leads to an especially efficient algorithm, which we call $PILU(1)$. This is because the $k = 1$ case requires no communication between nodes in the course of symbolic factorization. For the case $k = 2$ or larger k , we are still able to achieve a good speedup for symbolic factorization. The reason is that the density of the result matrix increases with larger k , and parallel $ILU(k)$ preconditioning for denser matrices requires more computation per computer node. This decreases the ratio of communication overhead to computation overhead, thereby producing better speedups. For numeric factorization, the speedup is always good because floating-point arithmetic forms a heavy burden for a single-CPU machine. That means the communication-computation ratio of numeric factorization is always smaller than that of symbolic factorization.

Therefore, the $k = 1$ case is the hardest case to achieve a good overall speedup. While we still present some experiments of $k \geq 2$ to show the features of the new parallel $ILU(k)$ algorithm, the results for $k = 1$ are the highlight. We demonstrate speedups approaching 40 in the best case on a typical departmental cluster. On a high performance cluster with an InfiniBand interconnect, we demonstrate nearly linear speedup even for 80 nodes, or in some cases 100 nodes. Our algorithm has the additional benefit of graceful degradation as communication delays in a wide-area network are introduced. Hence, the algorithm adapts well to the computational grid.

The paper is organized as follows. Section II briefly reviews some previous work. Section III reviews in-place LU factorization, sequential $ILU(k)$ algorithm and some implementation issues. Section IV presents the task-oriented parallel implementation TOP-ILU. Section V analyzes experimental results.

II. RELATED WORK

There are many sequential preconditioners based on incomplete LU factorization. Two typical sequential preconditioners are $ILUT$ and $ILU(k)$ [17]. A recent sequential preconditioner is $ILUC$ [20], [23]. In [19], a parallel version of $ILUT$ is given for distributed memory parallel computers. However, the parallelism in this paper comes from the analysis of a special non-zero pattern for a sparse matrix and does not have high scalability for a general sparse matrix.

In the process of parallelizing $ILU(k)$ preconditioner, we are faced with a natural problem: why is it so difficult to speed up $ILUT$ or $ILU(k)$ when k is small? We observe that $ILU(k)$ preconditioning is the kind of computation that accesses lots of memory while using relatively little floating-point arithmetic in the case of a huge sparse matrix of lower density with $k = 1$ or $k = 2$. Therefore, it is limited by either the memory bandwidth for the shared-memory case or the network bandwidth for the distributed-memory case when parallelizing and speeding up ILU preconditioner with more CPU s and CPU cores. Many discussions in [1], [6], [13] contribute valuable ideas that help us to handle this problem and design a scalable algorithm.

In [7], [8], [18], an LU factorization algorithm for distributed memory machines is implemented. However, this implementation needs a special API to update and synchronize the distributed memory. It implies that communication in the distributed memory model is a bottleneck even for LU factorization when huge sparse matrices are considered. So it is challenging to parallelize an $ILU(k)$ preconditioner on a cluster. However, it is important because cluster systems have become the mainstream of supercomputers. More than 70% of all supercomputers in the 2007 TOP500 list [28] are cluster systems.

In [21], the supernode data structure is used to reorganize a sparse matrix. Those supernodes can be processed in parallel. Observing that many rows have a similar non-zero pattern in the result matrix of LU factorization, rows with a similar non-zero pattern can be organized as one supernode.

The parallel ILU preconditioner in [27] aims toward distributed sparse matrices. $PMILU$ [14] presents a new technique to reorder a linear system in order to expose greater parallelism. They represent a category of parallel algorithms based on reordering. In [11], pivoting is employed to reduce the number of fill-ins for LU factorization. Similarly, pivoting is used to improve the robustness of the ILU preconditioner in [29]. The work in [5] provides an algorithm to make the diagonal elements of a sparse matrix large. The methodology is to compute the optimal pivoting and preprocess a sparse matrix. If the preprocessing makes the sparse matrix break-down free for $ILU(k)$ preconditioning, then it is possible to relax the diagonal dominance condition in our algorithm.

The paper [17] presents a graph-theoretic approach toward parallel $ILU(k)$ algorithm. The discussion there leads to a parallel algorithm for symbolic factorization, the first part of $ILU(k)$ preconditioning. However, there is no experimental result for their parallel $ILU(k)$ algorithm. The only other parallel $ILU(k)$ algorithm is in [15], [16]. This algorithm assumes that a matrix is well-partitionable [15] or that “it is possible to remove a small set of edges to divide the problem

into a collection of subproblems that have roughly equal computational work requirements” [16]. Because the algorithm acquires parallelism that depends on the special features of a matrix, this algorithm does not guarantee high scalability.

The task-oriented parallelism embodied by TOP-C [3] and introduced here describes a completely different approach from those mentioned above. The goal of the TOP-C philosophy is to parallelize sequential code while making minimal changes to the original sequential algorithm. The resulting parallel algorithm is derived by identifying tasks to be computed in parallel from the original sequential algorithm. TOP-C employs a message passing interface (MPI) for distributed memory architectures. It includes its own implementation of MPI, although it also allows a user to integrate TOP-C with any other MPI implementation such as MPICH2 [24] or OpenMPI [10], [25]. In [2], a definition of tasks suitable for Gaussian elimination is presented. A similar approach is applicable to LU factorization.

III. REVIEW OF SEQUENTIAL $ILU(k)$ ALGORITHM

For a more detailed review of $ILU(k)$, see [9], [17], [26]. This section provides a brief sketch. LU factorization completely decomposes a matrix A into the product of a lower triangular matrix L and an upper triangular matrix U . From matrices L and U , one efficiently computes A^{-1} as $U^{-1}L^{-1}$. While the computation of L and U requires $O(n^3)$ steps, once done, the computation of the inverse of the triangular matrices proceeds in only $O(n^2)$ steps.

For sparse matrices, one contents oneself with solving x in $Ax = b$ for vectors x and b , since A^{-1} , L and U would all be hopelessly dense. Iterative solvers are often used for this purpose. An $ILU(k)$ algorithm finds sparse approximations, $\tilde{L} \approx L$ and $\tilde{U} \approx U$. The iterative solver then implicitly solves $A\tilde{U}^{-1}\tilde{L}^{-1}$, which is close to the identity. This has faster convergence and better numerical stability. Here, the *level limit* k controls what kinds of elements should be computed in the process of incomplete LU factorization.

A. In-place LU Factorization

We begin by reviewing LU factorization. The goal is to express A as a product of L and U . $ILU(k)$ is a generalization of LU factorization. Suppose the matrix A is

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

Consider the transformation matrix L_1

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ -a_{21}/a_{11} & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ -a_{n1}/a_{11} & 0 & \dots & 1 \end{pmatrix}$$

The matrix L_1 transforms each row $R_i, i = 2, 3, \dots, n$ of A to the new row $R'_i = R_i - (a_{i1}/a_{11})R_1$. So the result matrix L_1A has the following form for new $a'_{ij}, i, j \geq 2$.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & a'_{n2} & \dots & a'_{nn} \end{pmatrix}$$

We recursively repeat the above process for the following submatrix of L_1A .

$$\begin{pmatrix} a'_{22} & \dots & a'_{2n} \\ \dots & \dots & \dots \\ a'_{n2} & \dots & a'_{nn} \end{pmatrix}$$

At the i^{th} recursive step, we obtain the following transformation matrix L_i .

$$\begin{pmatrix} 1 & \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ \dots & 1 & \dots & \dots & \dots \\ \dots & -a'_{i+1i}/a'_{ii} & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ \dots & -a'_{ni}/a'_{ii} & \dots & \dots & 1 \end{pmatrix}$$

It is easy to show that the result matrix $L_{n-1}L_{n-2} \cdots L_1A$ is exactly the *upper triangular matrix* U after all transformations are applied. In addition, the *lower triangular matrix* $L = (L_{n-1}L_{n-2} \cdots L_1)^{-1}$ has the form

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ a_{21}/a_{11} & 1 & \dots & 0 \\ a_{31}/a_{11} & a'_{32}/a'_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ a_{n1}/a_{11} & a'_{n2}/a'_{22} & \dots & 1 \end{pmatrix}$$

Therefore the above process is a method for LU factorization. This method can be implemented by the same procedure as Gaussian elimination. Moreover, it also records the elements of a lower triangular matrix L . So the LU factorization algorithm can be computed in place (without additional storage). The reason is that the row transformation starts from the current element and continues to the end. This does not affect elements that have previously been processed. Although this will destroy the original matrix A , it saves storage.

One wishes to store L and U in a single *filled matrix* F . For the off-diagonal elements, it is clear how to simultaneously store the triangular matrices L and U in F . For the diagonal elements, L is defined to be 1 on the diagonal, and so we store only the diagonal elements of U . The result matrix F then satisfies $f_{ij} = l_{ij}$ when $i > j$ and $f_{ij} = u_{ij}$ when $i \leq j$. Here f_{ij} , l_{ij} and u_{ij} are the elements of the matrices F , L and U respectively.

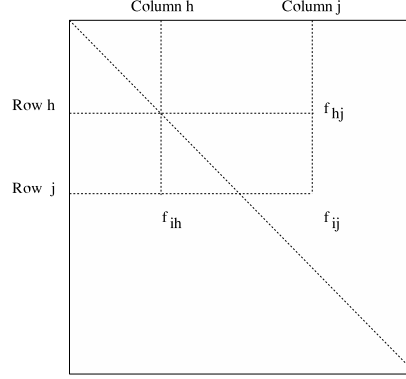


Fig. 1. Fill-in f_{ij} with its causative entries f_{ih} and f_{hj} .

B. Terminology for $ILU(k)$

For a huge sparse matrix, a standard dense format would be wasteful. Instead, we just store the position and the value of non-zero elements. Similarly, incomplete LU factorization does not insert all elements that are generated in the process of factorization. Instead, it employs some mechanisms to control how many elements are stored. $ILU(k)$ [17] uses the level limit k as the parameter to implement a more flexible mechanism. Here we review some definitions:

Definition 3.1: Filled Matrix: We call the matrix in memory the filled matrix, which is composed of all non-zero elements of F .

Definition 3.2: A fill entry, or entry for short, is an element stored in memory. (Elements that are not stored are called zero elements.)

Definition 3.3: Fill-in: Consider Figure 1. If there exists h such that $i, j > h$ and both f_{ih} and f_{hj} are entries, then the originally zero element f_{ij} may become an entry because the value of f_{ij} is non-zero after factorization. This element f_{ij} is called a fill-in, that is a candidate of entry. We say the fill-in f_{ij} is caused by the existence of the two entries f_{ih} and f_{hj} . The entries f_{ih} and f_{hj} are the causative entries of f_{ij} .

Definition 3.4: Level: The level associated with an entry f_{ij} , is denoted level (i, j) and defined as

$$\min_{1 \leq h < \min(i, j)} \text{level}(i, h) + \text{level}(h, j) + 1.$$

The level limit k is used to control what kinds of fill-ins should be inserted into the filled matrix during the factorization. Only those fill-ins with a level smaller than or equal to k are inserted into the filled matrix F . Other fill-ins are ignored. This allows $ILU(k)$ to maintain a sparse filled matrix for very small values of k .

Some versions of $ILU(k)$ use a *max rule* instead of the *sum rule* above. By the *max rule*, level (i, j) is defined as

$$\min_{1 \leq h < \min(i, j)} \max\{\text{level}(i, h), \text{level}(h, j)\} + 1.$$

Our method accepts either rule.

C. In-place Implementation of $ILU(k)$

We choose a row-major order for parallel $ILU(k)$ because the transformations in LU factorization lead to a natural parallel algorithm. Consider the transformation matrix L_1 as an example. All transformations for row $R_i, i = 2, 3, \dots, n$ can be done in parallel. In each transformation, two rows are accessed simultaneously from left to right.

For the reasons mentioned above, the sparse matrix is stored in row-major order form in our implementation. Hence, a cache-optimized algorithm will find that it is more efficient to access the matrix along rows, than along columns. There is a detailed discussion about the consideration of cache behavior for algorithm design in [4].

We next show that the row-major order leads to a natural in-place algorithm. The original matrix A is modified in place to represent \tilde{L} and \tilde{U} . Consider the computation for LU factorization. We expand the defining equation $A = LU$ into $a_{i,j} = \left(\sum_{k=1}^{j-1} l_{i,k}u_{k,j}\right) + f_{i,j}u_{j,j}$. This yields the defining equations.

$$\begin{aligned} f_{i,j} &= \left(a_{i,j} - \sum_{k=1}^{j-1} l_{i,k}u_{k,j}\right) / u_{j,j}, & j < i \\ f_{i,j} &= \left(a_{i,j} - \sum_{k=1}^{j-1} l_{i,k}u_{k,j}\right) / l_{i,i}, & j \geq i \end{aligned}$$

If the matrix is stored in row-major order form, the computation can be re-organized to use the above equations in the forward direction. Assume an entry of the F matrix has been computed. This corresponds either to $l_{i,k}$ or to $u_{k,j}$. Then that entry is used to modify a *later* entry of the matrix F . It is used to compute a single additional term in the summation that is required for future values of $f_{i,j}$.

We consider the case $j < i$ for specificity. Note that the equation $f_{i,j} = \left(a_{i,j} - \sum_{k=1}^{j-1} l_{i,k}u_{k,j}\right) / u_{j,j}$ makes it easy to accumulate these summation terms for the future value of $f_{i,j}$. The matrix F is initialized to A prior to any computation. As each term $l_{i,k}u_{k,j}$ for $k < j$ is determined, it can immediately be subtracted from $f_{i,j}$.

So, at any given time, the algorithm maintains in memory two rows: row i and row k , where $k \leq i$. Row i is used to *partially reduce* row k . In particular, for each possible j , the product $l_{i,k}u_{k,j}$ is used to reduce the entry $f_{i,j}$ of row i . By definition of U , $u_{k,j} = 0$ for $k > j$. So, once we have accumulated all products $l_{i,k}u_{k,j}$ for $k \leq j$, we are done.

The computation of f is graphically shown by Figure 2. The first column computes $f_{i,1}$. Similarly, the second column computes $f_{i,2}$, and so on. The value of $f_{i,j}$ is equal to the summation within column j of all items above the line in the middle and then by the item under the middle line. Because they are defined by induction, the computation should proceed from left to right.

The computation for incomplete LU factorization is similar to the above process except it skips zero elements. In fact, the program does not store the zero elements. Instead, it stores the column number of each non-zero entry.

D. Review of $ILU(k)$ Algorithm

According to the definition of $ILU(k)$, the implementation of the $ILU(k)$ algorithm includes two passes. In the first pass, we compute the levels and insert all fill-ins with the level less than

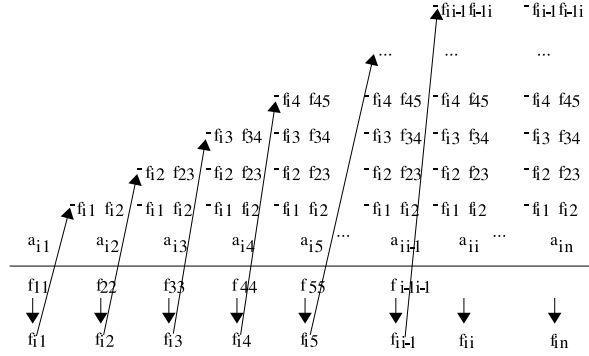


Fig. 2. The computation of the i^{th} row in F

or equal to the level limit k into the filled matrix. This pass is called *symbolic factorization* or Phase I. In the second pass, we compute the values of all entries in the filled matrix. This pass is called *numeric factorization* or Phase II.

The next algorithm shows the symbolic factorization phase. It determines for each row j , the set of permitted entries, $permitted(j)$. These are the entries for which the computed entry level or *weight* is less than or equal to the k in $ILU(k)$.

Numeric factorization is simpler, but similar in spirit to the row-merge update pass of Algorithm 1. The lines 14 through 17 control the entries to be updated, and the update of line 19 is replaced by the update formulas described in Section III-A. The details are omitted. The computation from step 15 to 27 in Algorithm 1 is referred to as one *transformation*. The corresponding part in Phase II is also called a transformation.

One difference between Phase I and Phase II is that the entry with the level attaining the level limit no longer contributes to the increase of any entry in Phase I. An optimization for this phase is that the entry is skipped if its level equals the level limit. This optimization makes symbolic factorization lightweight compared to numeric factorization, providing k is very small. A more detailed discussion is in Section V-C.

IV. TOP-ILU: TASK-ORIENTED PARALLEL $ILU(k)$ ALGORITHM

A. Task-oriented Parallelism in $ILU(k)$ Algorithm

As we have mentioned in section III-C, the parallelism comes from the fact that we can do row transformations in parallel while the sequential algorithm does it top-down and row by row. However, the computation for one row transformation is lightweight even for a large dense matrix. In order to overlap communication and computation, we increase the granularity of the computation. For this objective, the matrix is organized as bands. Just as in Figure 3, each band includes the same number of consecutive rows. To handle the case that the number of bands is not a factor of the matrix order, one can pad some of the bands with one extra row each. The size of a band is the number of rows in this band.

Consider Figure 3. After band 1 has been reduced completely, band 2, band 3 and band 4 are reduced simultaneously using the result of band 1. However, only band 2 can be reduced completely. Band 3 and band 4 are reduced partially. After band 2 is reduced completely, band 3

Algorithm 1 Symbolic factorization: Phase I of $ILU(k)$ preconditioning

```
1: //Calculate levels and permitted entry positions
2: //Loop over rows
3: for  $j = 1$  to  $n$  do
4:   //Initialization: admit entries in A, and assign them the level zero.
5:    $permitted(j) \leftarrow$  empty set //permitted entry in row  $j$ 
6:   for  $t = 1$  to  $n$  // nonzero entries in row  $j$  do
7:     if  $A_{j,t} \neq 0$  then
8:        $level(j, t) \leftarrow 0$ 
9:       insert  $t$  into  $permitted(j)$ 
10:    end if
11:  end for
12: end for
13: //Row-merge update pass
14: for each unprocessed  $i \in permitted(j)$  with  $i < j$ , in ascending order do
15:   for  $t \in permitted(i)$  with  $t > i$  do
16:      $weight = level(j, i) + level(i, t) + 1$ 
17:     if  $t \in permitted(j)$  then
18:       //already nonzero in  $F_{j,t}$ 
19:        $level(j, t) \leftarrow \min\{level(j, t), weight\}$ 
20:     else
21:       //zero in  $F_{j,t}$ 
22:       if  $weight \leq k$  //level control then
23:         insert  $t$  into  $permitted(j)$ 
24:          $level(j, t) \leftarrow weight$ 
25:       end if
26:     end if
27:   end for
28: end for
29: return  $permitted$ 
```

and band 4 can be reduced further with the result of band 2. Still, they are reduced in parallel. We regard as a single task each computation to reduce the band completely or partially. This algorithm is called TOP-ILU (task-oriented parallel ILU) because we acquire parallelism by identifying potentially parallel tasks in the sequential algorithm, following the approach of TOP-C [3].

B. Task-oriented Model For Matrix Row Reductions

Here we describe a general model for matrix row reductions, which is valid for Gaussian elimination as well as $ILU(k)$. (For $k = 1$, a different parallel model for symbolic factorization is used.)

Although the computation is organized as a bunch of tasks, the program needs to know how many rows have been reduced completely. So the following definition is introduced.

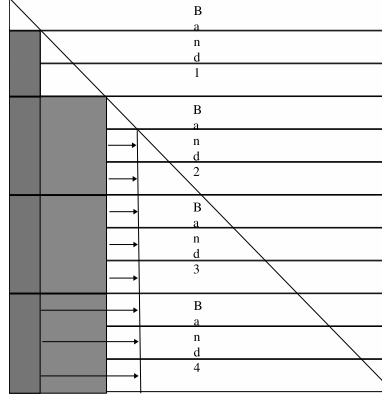


Fig. 3. View of a matrix as bands. There are four bands. Each of them consists of 3 rows. After the first band is reduced completely, all rest bands can be partially reduced to the third column in parallel.

Definition 4.1: *The frontier is the number of the last row such that it and all earlier rows are currently reduced completely. All later rows can be partially reduced such that for frontier i , the first i columns of the partially reduced row have their final value. (In the example of Gaussian elimination, that final value is always zero.) See Figure 3.*

A frontier i implies that all of the first i rows have been reduced completely. Meanwhile, the column i is the limit up to which the remaining rows can be partially reduced except for the first unreduced row. The first unreduced row can be reduced completely. That increases the *frontier* by one.

Regarding the matrix as a set of bands, each task is attached to one band. The task for a band represents a set of transformations applied in order to partially reduce the band to the current frontier. For each band, the program must remember up to what column this band has been partially reduced. We call it the *current position*, which is the start point of reduction for the next task attached to this band. In addition, it is important to use a variable to remember the first unreduced band. After the first unreduced band is completely reduced, the frontier should be increased by the size of the band on all machines.

As a “Worker” process completes its band, that band is broadcast to all other “Workers”. Each other “Worker” updates its copy of the matrix F accordingly, and uses the newly received band to update the band associated with the task assigned to that “Worker”.

The size of the band influences the size of the task in two respects. First, it determines the number of rows that should be processed in one task. Second, it determines the step size for advancing the frontier. So it is important to choose a suitable value of the band size. Too large a band size results in few bands and poor load balancing, as some “Workers” are starved for work. Too small a band size increases the communication costs. Because the computation costs for two passes of computation are quite different, our implementation allows one to use different band sizes for two passes, in order to make the computation and the communication commensurate individually in each pass.

C. TOP-ILU Algorithm

Our algorithm is described in terms of a single “Master” node and many “Worker” nodes, in accordance with TOP-C terminology. Initially, the matrix to be solved is initialized by the “Master” and all “Workers” respectively. When generating a task, the “Master” scans all unreduced bands and finds the first band that has not been reduced by comparing the current position with the frontier. If found, the “Master” sends the band number to an idle “Worker”. Then the “Master” continues to check the status of “Workers”. In our algorithm, the task ID is just the band number because all machines know the current position of each band and the frontier.

After the idle “Worker” receives the task, it begins to do the task. That task is to reduce all rows in the band by the frontier band. When the task is finished, the “Worker” sends the result to the “Master”. Then the “Master” forwards the result to all “Workers” in a special update message. Both the “Master” and “Workers” should update their own copy of the filled matrix, the current position of the updated band and the current value of the frontier if necessary. After the update message has been handled, all machines should have the same image of the filled matrix. This mechanism is important for dynamic load balancing.

D. Performance Estimation and Improvement

Analyzing the communication model of master-worker architecture, we know that each time the worker reduced one band to the frontier, the whole set of data in this band must be first submitted to the “Master” and then forwarded to all other “Workers” by the “Master”. The benefit is that the following reduction of this band can be handled by any idle “Worker”. This implements perfect load balancing. The price is that all intermediate results must broadcast to all other “Workers”. We call this *dynamic load-balancing*.

One observation concerning our algorithm is that only the completely reduced bands are useful for the reduction of other bands. So the intermediate result is not truly needed by other “Workers”. If we ensure that all machines handle the fixed group of bands, then it is unnecessary to broadcast the update message for any intermediate result to other “Workers”. We call this strategy *static load balancing*. It decreases the communication overhead to a minimum by sending the update message only for the completely reduced band.

Our algorithm assigns bands to each node in a round-robin fashion. The method avoids sending intermediate copies of partially updated bands. This produces a more regular communication that fits well with the pipelining communication of the next section. A further virtue of this strategy is that it uses a fixed number of message buffers and a fixed buffer size. This avoids dynamically allocating the memory for message handling. Under the strategy of static load balancing, the computations on all processors are coordinated so as to guarantee that no processor can send two update messages simultaneously. In other words, before one processor finishes broadcasting an update message, it is impossible for this processor to reduce another band completely.

E. Pipeline Communication for Efficient Local Network Usage

In order to maintain the same current copy of the matrix, each completely reduced band must be sent to all other machines. For our parallel $ILU(k)$ algorithm, although the bands can be reduced simultaneously, they are reduced completely following the strict top-down order. When

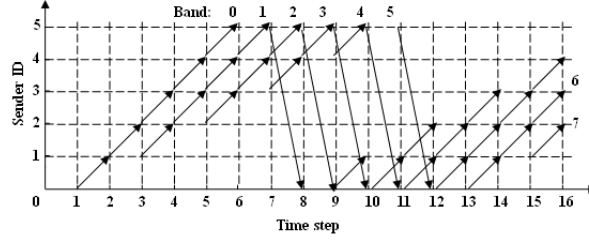


Fig. 4. “Pipeline” model. The horizontal axis is the time step. The vertical axis is the sender id. The lines represent when the algorithm sends a message. The time step and the sender id of the source are indicated. The receiver is always the successor of the source. The message is marked by the corresponding band number. Only the first several messages are shown.

one band is completely reduced, it is better for the node that reduces the next band to obtain the result first. This can be implemented by the “pipeline” model. Our study, based on experiments, shows that this model is the optimum for the parallel $ILLU(k)$ algorithm using up to one hundred CPUs.

Following this model, all nodes are organized to form a directed ring. The message is transferred along the directed edge. Every node sends the message to its unique successor until every node has received a copy of the message. After this message is forwarded, each node uses this message to update the memory data.

We use Figure 4 to illustrate how this model achieves the aggregate bandwidth. In this figure, the horizontal axis is the time step while the vertical axis is the sender ID (the rank number of each node). Note that at a time step when all nodes are participating, then each node is either sending a message to its successor or receiving a message from its predecessor.

Algorithm 2 shows how the “pipeline” model is integrated with the computation. Both symbolic factorization and numeric factorization can use this algorithm. However, for the case of $k = 1$, symbolic factorization reduces to trivial parallelism as discussed in the next section.

F. $PILU(1)$ Algorithm: A Special Case for Symbolic Factorization in The Case $k = 1$

In the $k = 1$ case, we use more CPUs to obtain a good speedup for numeric factorization. But there would be no corresponding speedup for symbolic factorization in this case. Luckily, there is a different parallel approach for efficiently implementing symbolic factorization for the case $k = 1$. We call this special algorithm $PILU(1)$. The following two paragraphs explain the $PILU(1)$ algorithm.

One observation from Figure 1 is that level 1 entries no longer participate in symbolic factorization after they are generated. In Figure 1, if either f_{ih} or f_{hj} is the entry of level 1, the caused fill-in f_{ij} must be the element of level 2 or level 3. So f_{ij} is not inserted into the filled matrix F . By this observation, we claim that each row can be reduced independently no matter whether the previous rows are reduced or not. This observation not only yields greater parallelism, it also allows the communication overhead of the first pass to be postponed to the second pass. The “Master” and “Workers” do not synchronize the filled matrix in the first pass. However, both column number and value are sent in the second pass. Considering that numeric factorization is floating-point arithmetic intensive, it is reasonable to shift all communication overhead toward the second pass.

Algorithm 2 Parallel $ILU(k)$ algorithm with the “pipeline” model

```
1: receive from predecessor //non-blocking receive
2: //Loop until all bands are reduced completely
3: while firstUnreducedBand < numberOfBands do
4:   get new task (band  $ID$ ) from the “Master” to work on
5:   if there was a band to work on then
6:     doTask(band) // reduce band using all previous bands
7:     if band is not reduced completely then
8:       //not reduced completely, then non-blocking test
9:       try to receive a message for some band
10:    if a newly reduced band is received then
11:      send band to successor //non-blocking send
12:      update our copy of newly reduced band
13:      continue to receive and update until our band is completely reduced
14:    end if
15:  else
16:    send our reduced band to successor //non-blocking send
17:  end if
18: else
19:   wait until a new band is available, while in background continuing to receive other
   reduced bands from predecessor, updating our copy, and sending the reduced band to
   our successor
20: end if
21: end while
```

In the symbolic factorization phase, the $PILU(1)$ algorithm prefers to reduce each row with the original matrix because only level 0 elements are needed. In addition, the algorithm achieves better performance if it does not scan the new inserted level 1 elements. This optimization can be implemented by letting all “Workers” reduce rows bottom-up. Meanwhile no update is sent to any other processor. Under such circumstances, the band size for symbolic factorization does not influence the performance because there is no synchronization among processors. However, we must use the same band size for symbolic factorization as for numeric factorization.

V. EXPERIMENTAL RESULTS

We use sparse matrices generated by matgen [22] in most experiments. The matrices are stored in a sparse representation: each matrix is an array of rows, each of them is an array of entries. Each entry contains a column number and level (as required by $ILU(k)$), and a matrix value. They are stored as two integers and a float number in program. A copy of the filled matrix is assumed to reside on each node of the cluster before the computation. After the computation, the result matrix also resides on each node of the cluster.

A. Dynamic Load Balancing vs. Static Load Balancing

First we compare dynamic load balancing with static load balancing. For this purpose, we used an older cluster. The older cluster is less in demand, and so it is easier to find lightly loaded machines. This cluster has 65 nodes connected by Gigabit Ethernet. Each node consists of two processors of Intel Xeon running at 3.2 GHz and 4 GB of RAM. The version of kernel is Linux 2.4.x. We use MPICH2.

The speedup of both methods on matrix of dimension 20,000 is collected for the cases of 4, 7 and 10 CPUs respectively. The results are in Table I. In this table, column LB denotes load balancing and S speedup. In column LB, value D denotes dynamic load balancing and S static load balancing.

n	LB	#CPU	k	#Band	Time	S
20K		1	2		37.9	
20K	D	4	2	30	19.5	1.9
20K	S	4	2	1024	11.8	3.2
20K		1	3		1277.4	
20K	D	7	3	160	274.5	4.7
20K	S	7	3	1024	203.0	6.3
20K	D	10	3	160	187.9	6.9
20K	S	10	3	10240	132.4	9.6

TABLE I
DIFFERENT METHODS OF LOAD BALANCING

Table I shows that the computation of the improved algorithm in Section IV-D is stable even for very large band numbers. This implies that static load balancing obtains greater parallelism and achieves greater speedup.

Second, we use a departmental cluster to study the features of our $ILU(k)$ algorithm. This cluster consists of 33 servers, all having 4 CPU cores (dual processor, dual-core), 2.0 GHz Intel Xeon EM64T processors with both 8 GB and 16 GB memory configurations. All servers are connected by a Gigabit Ethernet network. The version of kernel is Linux 2.6.x. We still use MPICH2. At most two of the four cores are used on each node.

B. Driven Cavity Problem $e40r3000$

Here we introduce a real-world example from the *SPARSKIT* Collection [15]. The problem Driven Cavity $e40r3000$ is a 17281×17281 matrix with 553956 entries. It arises from the modeling of the incompressible Navier-Stokes equations. This problem is solved using the sequential $ILU(k)$ algorithm proposed in the paper for cases $k \geq 2$ in [15]. That paper also proposes a parallel $ILU(k)$ algorithm. But this algorithm assumes that the matrix is well-partitionable. So it does not always work. There is not any experimental result of parallel preconditioning for this real-world example.

The paper shows that the faster solution time for $e40r3000$ was obtained with sequential $ILU(3)$ preconditioning even though sequential $ILU(6)$ preconditioning achieves the least number of iterations. The reason that the sequential $ILU(6)$ fails to obtain the best solution time is that the preconditioning occupies 86% of the total computation.

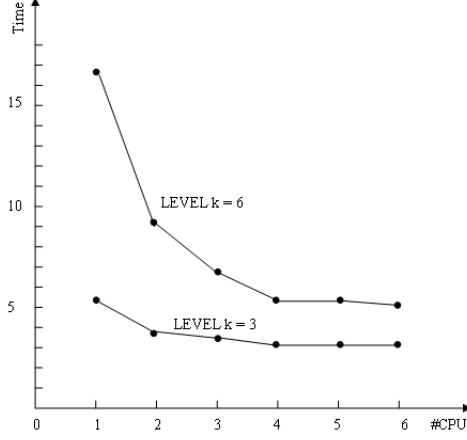


Fig. 5. Times for $ILU(3)$ and $ILU(6)$ preconditioning for the matrix e40r3000.

We use our algorithm to compute both the parallel $ILU(3)$ preconditioner and the parallel $ILU(6)$ preconditioner. The results are in Figure 5. From this figure, we can see that the computation for the $ILU(6)$ preconditioner is far greater than that for the $ILU(3)$ preconditioner when the sequential algorithm is used. However, using task-oriented parallelism with 6 CPU s, they obtain a speedup and finish in 3.1s and 5.2s respectively. In addition, the result matrix of the parallel $ILU(k)$ preconditioning is equal to the result matrix of the sequential $ILU(k)$ preconditioning except the parallel preconditioning consumes less time. Considering that the $ILU(6)$ preconditioner decreases the number of iterations greatly comparing with the $ILU(3)$ preconditioner, it is expected that we achieve better performance using the parallel $ILU(6)$ preconditioner.

C. Symbolic Factorization vs. Numeric Factorization

In this group of experiments, we figure out how the ratio of symbolic factorization to numeric factorization changes when k increases gradually. The results are in Figure 6. We use four matrices to measure the computation time for both symbolic factorization and numeric factorization for cases $k = 1$ up to 5. The matrix densities are 0.073, 0.036, 0.009 and 0.002 respectively. In all cases, the ratio of symbolic factorization to numeric factorization does not decrease when we increase k gradually. If k is large enough, the ratio goes beyond 1.

Therefore we claim that the time for symbolic factorization is almost the same as or even a little greater than that for numeric factorization if no entry is skipped in the first phase. Although the non-zero elements are inserted dynamically and this results in fewer comparisons, the insertion of an entry is costly. To insert an entry, we move the remaining entries in order to open enough space for the new fill entry.

In practical applications with small k , symbolic factorization is lightweight due to the optimization in Section III-D. Consider the matrix of dimension $20K$ used in Section V-A. For the case $k = 1$, the number of entries is 1,239,058 after symbolic factorization. All those entries are involved in numeric factorization. On the other hand, only 265,563 level 0 elements cause new entries in the symbolic factorization phase.

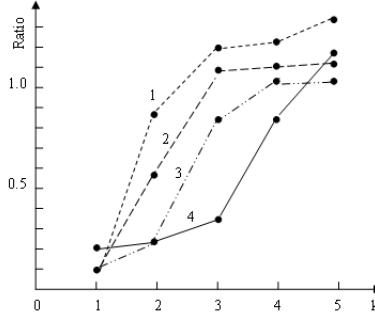


Fig. 6. Comparison of symbolic factorization and numeric factorization (sequential algorithm); LEVEL $k = 1, 2, 3, 4$ and 5 ; Curve 1 is the result for a 1024×1024 matrix; Curve 2 is the result for a 2048×2048 matrix; Curve 3 is the result for a 4096×4096 matrix; Curve 4 is the result for a 8192×8192 matrix.

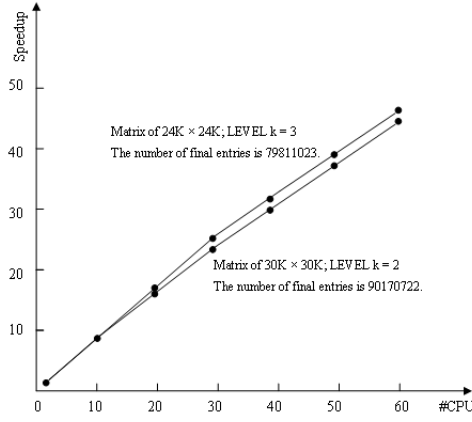


Fig. 7. Computation for higher level; The matrix densities are 0.00061 and 0.00089.

D. Parallel $ILU(k)$ Algorithm for Larger k

We next study the speedup of the algorithm for larger k . For the situations $k = 2$ and $k = 3$, the speedup is good, as demonstrated by the experimental results in Figure 7. For the $24K$ matrix with initial density 0.00061, $k = 3$ creates 79811023 final entries. For the $30K$ matrix with initial density 0.00089, $k = 2$ creates 90170722 final entries. For both situations, we achieve nearly linear speedup through the largest example: 60 CPUs.

The result of nearly linear speedup for more CPUs is reasonable, since raising k to 2 or 3 causes more fill-ins, and a denser result matrix. The added floating-point arithmetic per task implies a larger computation-communication ratio.

E. Case $k = 1$: $PILU(1)$ Algorithm

It is not difficult to speed up the computation for cases with larger k . Therefore, in the following experiments, we primarily study the case $k = 1$, which is the most commonly used case. The first group of experiments is executed on the departmental cluster. The results of the sequential $ILU(k)$ algorithm are in Table II. The results of the $PILU(1)$ algorithm are in Table III. In this

table, column S denotes speedup, as in the previous table. For $k = 1$, all causative entries are initial entries with level 0. So in the first pass, merely initial entries contribute to increasing the size of the filled matrix and updating levels for new inserted entries. There is no communication in the first pass. The final number of entries determines the communication overhead of a single machine for the second pass. The final entry number, density and non-zero pattern determine the amount of computation in the second pass.

n	#Initial entry	#Final entry	Time
40K	5120950	196223519	445.2 + 8938.2
80K	6960983	195202037	234.0 + 4120.8
160K	9832794	198969083	140.2 + 2112.7
320K	14090553	206489590	93.4 + 1162.0

TABLE II
COMPUTATION OF SEQUENTIAL ALGORITHM; LEVEL $k = 1$; THE MATRIX DENSITIES ARE 0.003, 0.001, 0.00037 AND 0.00013.

n	#CPU	#Band	Time	S
40K	50	20480	9.5 + 217.4	41.4
40K	60	20480	7.8 + 191.6	47.1
80K	40	20480	6.7 + 142.9	29.1
80K	60	20480	5.0 + 98.7	42.0
160K	30	40960	4.7 + 101.8	21.2
160K	60	40960	2.5 + 64.4	33.7
320K	30	81920	3.3 + 89.8	13.5
320K	40	81920	2.4 + 71.3	17.0
320K	60	81920	1.6 + 59.0	20.7

TABLE III
COMPUTATION ON THE DEPARTMENTAL CLUSTER; LEVEL $k = 1$; THE MATRIX DENSITIES ARE 0.003, 0.001, 0.00037 AND 0.00013.

From Table II and Table III, we can see the following. For a matrix of dimension 40,000 with density 0.003 and a matrix of dimension 80,000 with density 0.001, we obtain a linear speedup for 60 CPUs. For a matrix of dimension 160,000 with density 0.00037, and a matrix of dimension 320,000 with density 0.00013, we obtain the maximal speedup at 60 CPUs.

The symbolic factorization phase always obtains a linear speedup because there is no communication overhead. However, numeric factorization phase does not achieve the linear speed for all cases. The decreasing speedup in the phase of numeric factorization, which is the major part that influences the total speedup, is explained as follows.

Suppose the matrix has n_f entries finally. By “pipeline” model, the communication overhead is about $8n_fB$ per node. The reason is that both the column number and the value of entries in those bands that are not handled by the successor are sent to the successor.

Considering four matrices in above experiments, the numbers of final entries are all $200M$. So the communication overhead is about $200M \times 8B$ per node in the second phase. As the matrix dimension increases, the density decreases, as does the amount of floating-point arithmetic and computation-communication ratio. Also the optimal number of CPUs then decreases.

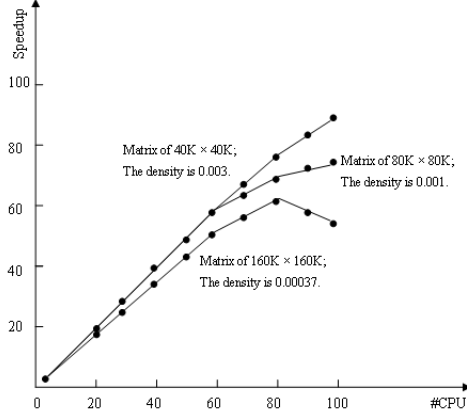


Fig. 8. Computation on Lonestar; LEVEL $k = 1$.

Increasing the number of CPUs decreases the computation-communication ratio by increasing the total communication overhead and decreasing the computation burden of each machine. In order to improve the speedup further, we must decrease the total communication time to ensure the communication and the computation overlap well.

To increase bandwidth is one solution. We use a high performance cluster lonestar.tacc.utexas.edu with more bandwidth to repeat experiments for the matrices of 40K, 80K and 160K. The Lonestar cluster is configured with 5200 compute-node processors connected by Gigabyte network. Each node contains two Xeon Intel Duo-Core 64-bit processors (4 cores in all) and 8 GB memory. The Core frequency is 2.66GHz. The version of kernel is Linux 2.6. We use MPI MVAPICH (mvapich) and Intel 9.1 compiler (intel) that exist in the default environment.

Figure 8 is experimental results on Lonestar. It shows that our algorithm has the scalability to 80–100 CPUs given sufficient bandwidth.

F. Scalability for The Grid

Sometimes the computation is heavy for a departmental cluster usually with 60 CPUs. To take advantage of the computation capacity of a remote cluster, we encounter the high latency of inter-cluster. The intra-cluster latency is usually about a few μs while the inter-cluster latency a few ms . To move our algorithm to the Grid, this latency must be considered.

In our tests on the Internet, the round-trip time is always less than 35ms (17.5ms one way). So we simulate the communication latency of the Grid on cluster by adding some delay above 17.5ms before sending a message for each edge node (a node communicating between clusters). We generate a $32K \times 32K$ matrix with the initial density 0.00458 to perform experiments. The number of final entries increases to 210,212,433 after symbolic factorization.

The results are in Figure 9. In this figure, the number of CPUs is expressed as the number of clusters times the number of CPUs of each cluster. From this figure, we see the following.

First, the average bandwidth of communication on the Grid is not as good as that on a cluster due to the communication latency. This always makes the speedup decrease. The maximal speedup in Figure 9 occurs when the number of *CPUs* is 100, when the simulated latency is zero.

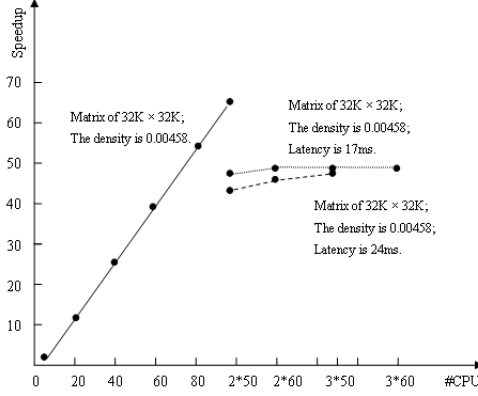


Fig. 9. Simulation of the performance on the Grid; LEVEL $k = 1$.

Second, more edge nodes result in more communication time. To increase the number of clusters does not necessarily contribute to more speedup because of increased number of edge nodes. In the case of 17ms latency, 2×60 CPUs, 3×50 CPUs and 3×60 CPUs achieve a similar speedup.

Third, the speedup decreases as the latency increases. The algorithm performs better with 17ms latency than with 24ms latency, assuming other parameters are fixed.

Fourth, the performance of our algorithm is not good when the *CPUs* are distributed among more clusters. However, the performance deteriorates very little for both the 17ms latency case with 3 clusters and the 24ms latency case with 2 clusters.

In practical applications, the performance should be better than the result in our experiments because our implementation allows part of latency to be hidden by the computation in the second pass. In addition, there is no communication in the first pass.

One strategy to mitigate the influence of latency is to use as small a number of bands as possible, given that there are still enough bands for balancing the computation and the communication.

VI. CONCLUSIONS

For $k = 2$ and $k = 3$, we achieve nearly linear speedup because computation dominates communication as we use more floating-point arithmetic per task. For $k = 1$, we use a modified version of our parallel algorithm, *PILU*(1). This algorithm produces a 21-fold speedup on a departmental cluster (Gigabit Ethernet) over 30 nodes, operating on a matrix of dimension 160,000 and density 0.00037. On a high performance cluster (InfiniBand interconnect), we demonstrate a 58-fold speedup with 60 nodes operating on a matrix of dimension 80,000 and density 0.001.

We also simulated a Grid computation with two and three clusters participating in a computation. A communication delay was added for all messages passing between distinct simulated clusters. We tested on a matrix of dimension 32,000 and density 0.005. A single cluster of 100 nodes exhibited 64 times speedup. Two clusters of 50 nodes each caused the speedup to degrade to only 45 times (for a typical 17 ms delay over the Internet), and to a 42 times speedup

for a 24 ms long delay. Adding a third cluster of 50 nodes contributed almost no additional speedup.

Our parallel $ILU(k)$ algorithm is fully general and depends only on a standard diagonal dominance condition. This condition is typically assumed even in descriptions of the sequential $ILU(k)$ algorithm. Moreover, our parallel $ILU(k)$ algorithm produces the same answer as the sequential $ILU(k)$ algorithm. Some other parallel $ILU(k)$ algorithms such as graph decomposition [14], [15], [16] change the row or column order, and so they cannot make this guarantee.

The diagonal dominance is critical for $ILU(k)$ preconditioner for two reasons. First, the definition of level does not allow one to change the order of rows or columns. This fixed ordering plus diagonal dominance then ensures that diagonal entries remain non-zero. Second, if the matrix is diagonally dominant, the higher the level of a fill-in, the smaller the fill-in in absolute value. With this condition, we believe that a good approximation of L and U can still be achieved by ignoring those high level elements.

Early graph-decomposition methods acquired parallelism that was highly dependent on the matrix structure. We emphasize a general parallel algorithm. Two methods complement each other. In future work, we will follow the TOP-C philosophy to raise the speedup based on a TOP-C shared memory thread model.

VII. ACKNOWLEDGEMENT

We acknowledge helpful discussions with Ye Wang at an early stage of this work.

REFERENCES

- [1] M. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182:418–477, 2002.
- [2] G. Cooperman. Practical task-oriented parallelism for Gaussian elimination in distributed memory. *Linear Algebra and its Applications*, 275–276:107–120, 1998.
- [3] G. Cooperman. TOP-C: A library that links with your existing sequential code (after small modifications) in order to parallelize it, 2003. <http://www.ccs.neu.edu/home/gene/topc.html>.
- [4] G. Cooperman and X. Ma. Overcoming the memory wall in symbolic algebra: A faster permutation algorithm. *SIGSAM Bulletin*, 36:1–4, 2002.
- [5] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.
- [6] I. S. Duff and H. A. van der Vorst. Developments and trends in the parallel solution of linear systems. *Parallel Computing*, 25:1931–1971, 1999.
- [7] C. Fu, X. Jiao, and T. Yang. Efficient sparse LU factorization with partial pivoting on distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, 9, 1998.
- [8] C. Fu and T. Yang. Sparse LU factorization with partial pivoting on distributed memory machines. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CD-ROM)*, volume 31, 1996.
- [9] G. Golub and C. V. Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.
- [10] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.
- [11] L. Grigori, J. Demmel, and X. S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM J. Scientific Computing*, 29(3):1289–1314, 2007.
- [12] I. Gustafsson. A class of first order factorization methods. *BIT Numerical Mathematics*, 18:142, 1978.
- [13] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–240, 1991.
- [14] P. Hénon and Y. Saad. A parallel multistage ILU factorization based on a hierarchical graph decomposition. *SIAM J. Scientific Computing*, 28:2266–2293, 2006.
- [15] D. Hysom and A. Pothen. Efficient parallel computation of $ILU(k)$ preconditioners. Tech Report 2000-23, ICASE, NASA Langley Research Center, 2000.

- [16] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning, 2000.
- [17] D. Hysom and A. Pothen. Level-based incomplete LU factorization: Graph model and algorithms. Tech Report UCRL-JC-150789, Lawrence Livermore National Labs, 19 pages, 2002.
- [18] B. Jiang, S. Richman, K. Shen, and T. Yang. Efficient sparse LU factorization with lazy space allocation. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.
- [19] G. Karypis and V. Kumar. Parallel threshold-based ILU factorization. Technical Report 061, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 5455, 1998.
- [20] N. Li and Y. Saad. Crout versions of the ILU factorization with pivoting for sparse symmetric matrices. *Electronic Transactions on Numerical Analysis*, 20:75–85, 2005.
- [21] X. Li. *Sparse Gaussian Elimination on High Performance Computers*. PhD thesis, Computer Science Division, EECS, U. of California, Berkeley, 1996.
- [22] Matgen: Command line random matrix generator on Linux. <http://matgen.sourceforge.net/>.
- [23] J. Mayer. Some new developments in ILU preconditioners. In *GAMM Annual Meeting*, volume 6, pages 719–720, 2006.
- [24] For many important platforms. <http://www-unix.mcs.anl.gov/mpi/mpich2>.
- [25] A high performance message passing library. <http://www.open-mpi.org>.
- [26] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20th century. *J. Comput. Appl. Math.*, 123:1, 2000.
- [27] C. Shen, J. Zhang, and K. Wang. Parallel multilevel block ILU preconditioning techniques for large sparse linear systems. In *Proceedings of International Parallel and Distributed Processing Symposium*, pages 22–26, 2003.
- [28] Top500list. <http://www.top500.org>, 2007.
- [29] J. Zhang. A multilevel dual reordering strategy for robust incomplete LU factorization of indefinite matrices. 22:925, 2001.